Austin Brown

CPE 434-01

1/28/2021

Lab 3

**Theory**

In this lab we are implementing out own Linux shell. A shell is a program that takes user input from a keyboard, parses it, and gives to the operating system to perform. The shell that we are implementing is a Bash shell. There are several important functions that we will use. They include strtok, dup, dup2, pipe, execvp.

The strtok function splits a string into parts or tokens based on a delimiter. These strings can be stored in an array. This is a 2d array since strings are arrays of character, and we are creating an array of strings. The delimiter used is space. You have the command followed by arguments and possibly other commands if a pipe is used.

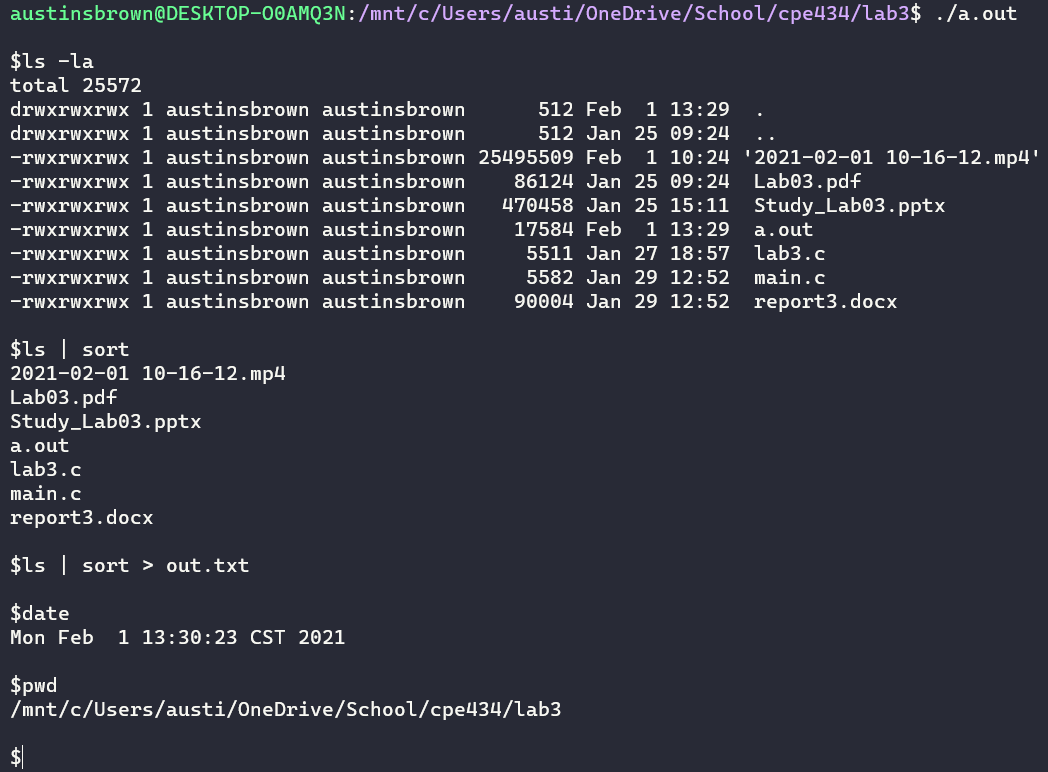
The dup and dup2 functions are used to create copies of file descriptors. A file descriptor is an integer that identifies an open file. It returns negative 1 there is an error. The difference between dup and dup2 is that dup assigns the lowest available file descriptor. Dup2 lets you choose what file descriptor you want. It automatically closes the old one if your chosen descriptor is taken.

The pipe function allows us to send information from one process to another. The function takes a 2-element array as input. The first element is the file descriptor for the left or read end of the pipe. The second element is the file descriptor for the right or write end of the pipe. Pipes act as a queue or first in first out data structure.

The execvp function is used to execute files. The first argument is the file to be executed. The second argument is an array of strings that represent the possible arguments to the command being executed. This function is used to execute the user input in my shell.

**Observations**

The output for several commands is shown below.

****

**Conclusion**

Writing this program was a pain but I learned a lot from it. In previous labs we learned how to create new processes. In this lab, we take that a set further and pass data from one process to the other using the pipe function. We also learned about the execvp, strtok, dup, and dup2 functions.

**Appendix**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <fcntl.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  #define BUFFER\_SIZE 255  *int* main()  {        for(;;)      {  *char* \*buffer = malloc((BUFFER\_SIZE)\*sizeof(*char*)); *// holds the command eneterd by the user*          printf("\n$");          fgets(buffer, BUFFER\_SIZE, stdin); *// get the input*          buffer[strcspn(buffer, "\n")] = 0; *// remove the trailing newline character*  *char* \*\*stringArray = malloc((BUFFER\_SIZE)\*sizeof(*char*\*)); *// allocate a 2d array to store the stringArrayized command*          for (*int* i = 0; i < (BUFFER\_SIZE); i++)              stringArray[i] = malloc((BUFFER\_SIZE)\*sizeof(*char*));  */\* 1: right redirect*  *2: left redirect \*/*  *int* \*redirect = malloc(sizeof(*int*));  *int* \*redirectDestination = malloc(sizeof(*int*));  *int* \*pipeFlag = malloc(sizeof(*int*)); *// couts the number of pipes*  *int* argCount = 0; *// generic counter*          \*pipeFlag = 0; *// counts number of pipes*          \*redirect = 0;          stringArray[0] = strtok(buffer, " "); *// begin breaking up the string*          while (stringArray[argCount] != NULL)          {              stringArray[++argCount] = strtok(NULL, " ");              if (stringArray[argCount] != NULL && strcmp(stringArray[argCount], ">") == 0) *// handle right redirect*              {                  \*redirect = 1;                  stringArray[argCount] = strtok(NULL, " ");                  \*redirectDestination = open(stringArray[argCount], O\_CREAT|O\_RDWR|O\_TRUNC, 0644); *// create a file with read write permissions and truncate to 0*                  argCount--;              }              else if ((stringArray[argCount] != NULL) && (strcmp(stringArray[argCount], "<") == 0)) *// handle left redirect*              {                  \*redirect = 2;                  stringArray[argCount] = strtok(NULL, " ");                  \*redirectDestination = open(stringArray[argCount], O\_CREAT|O\_RDWR|O\_TRUNC, 0644); *// create a file with read write permissions and truncate to 0*                  argCount--;              }              else if ((stringArray[argCount] != NULL) && (strcmp(stringArray[argCount], "|") == 0)) *// handle pipes*                  \*pipeFlag = argCount;          }          stringArray[argCount] = NULL;          if (stringArray[0] != NULL && strcasecmp(stringArray[0], "exit") == 0)      {          printf("Goodbye\n");          exit(0);      }      if (\*pipeFlag == 0)      {  *pid\_t* pid = fork();          if (pid > 0) *// parent waits for the child*              wait(0);            else if (pid == 0) *// child executes*          {              if (\*redirect == 0)                  execvp(stringArray[0], stringArray);                else if (\*redirect == 1 && \*redirectDestination != -1) *// if right redirect and an error has not occured*              {                  dup2(\*redirectDestination, 1);                  execvp(stringArray[0], stringArray);              }              else if (\*redirect == 2 && \*redirectDestination != -1) *// if left redirect and an error has not occured*              {                  dup2(\*redirectDestination, 0);                  execvp(stringArray[0], stringArray);              }              else if (\*redirectDestination == -1) *// if an error has occured*              {                  printf("Could not create file.\n");                  exit(-1);              }              exit(0);          }          else          {              printf("Error! Could not create child.\n");              exit(-1);          }      }      else      {  */\* pipeDes[0]: file descriptor for the read end of pipe*  *pipeDes[1]: file descriptor for the write end \*/*  *int* pipeDes[2];          if (pipe(pipeDes) == -1)              printf("Pipe Ded\n");          else          {  *char*\* leftProgram[BUFFER\_SIZE]; *// holds the left end of the pipe*  *char*\* rightProgram[BUFFER\_SIZE]; *// holds the right end of the pipe*  *int* i;  *int* j = 0;              for(i = 0; i < argCount; i++) *// populate the left and right program buffers*              {                  if (i < \*pipeFlag) *// populate the left buffer*                      leftProgram[j++] = stringArray[i];                    else if (i == \*pipeFlag)                  {                      leftProgram[j] = NULL;                      j = 0;                  }                  else if (i > \*pipeFlag) *// populate right program*                  {                      leftProgram[j++] = stringArray[i];                  }              }              rightProgram[j] = NULL;  *int* pid = fork();              if (pid == 0) *// execute the left program*              {                  dup2(pipeDes[1], 1);                  close(pipeDes[0]);                  execvp(leftProgram[0], leftProgram);                  printf("Error 1\n");                  exit(1);              }              pid = fork();              if (pid == 0) *// execute the right program*              {                  dup2(pipeDes[0], 0);                  close(pipeDes[1]);                  if (\*redirect > 0)                      dup2(\*redirectDestination, 1);                  execvp(rightProgram[0], rightProgram);                  printf("Error 2\n");                  exit(1);              }              close(pipeDes[1]);              close(pipeDes[0]);              wait(0);              wait(0);          }      }          free(buffer);          free(stringArray);          free(redirect);          free(redirectDestination);          free(pipeFlag);      }      return 0;  } |